Chapter 7
Compression and Pattern Matching

Takuya Kida and Isamu Furuya

Abstract We introduce our research on compressed pattern matching technology that combines data compression and pattern matching. To show the results of this work, we explain the collage system proposed by Kida et al. in 2003 that is a unifying framework for compressed pattern matching, and we explain the Repair-VF method proposed by Yoshida and Kida in 2013 and the MR-Repair method proposed by Furuya et al. in 2019 as grammar compressions suitable for compressed pattern matching.

7.1 Introduction

Data compression is a technology that reduces the space used to store data by compactly expressing the redundancy contained in the data. It is mainly used for efficiently storing large amounts of data and reducing communication costs. If we consider the conversion of information to digital data as a kind of data compression, it has a long history that can be traced back to the Morse code developed in the 1830s. Many compression methods have been proposed depending on the type and application of data [43–46].

Information retrieval has also long been studied as a technique for efficiently finding a target part from a large-scale dataset or data group [3, 11, 13, 14, 30, 39, 55], and there are various methods depending on the required specifications. In particular, the approaches differ between searching for images and audio data and searching for documents (*text*). In this chapter, we focus on the latter task of text searching.

T. Kida (✉)
Faculty of Engineering, Hokkai-Gakuen University, 1-1, S26-W11, Chuo-ku, Sapporo 064-0926, Japan
e-mail: kida@lst.hokkai-s-u.ac.jp

I. Furuya
Graduate School of IST, Hokkaido University, N14-W9, Kitaku, Sapporo 060-0814, Japan
e-mail: furuya@ist.hokudai.ac.jp

N. Katoh et al. (eds.), *Sublinear Computation Paradigm*,
https://doi.org/10.1007/978-981-16-4095-7_7

One of the basic problems in text searching is the *pattern matching problem*, which is also called the string matching problem. This is the problem of finding the occurrences of keywords (*patterns*) in a target text. Broadly speaking, there are two approaches to solving this problem. One is to build an index for the input text in advance, which is called *text indexing*. A text index allows for efficient searching when the target text is static and is not subsequently updated. The other is to access the input text sequentially from the beginning to the end while checking if the given pattern matches at the current reference position in the text. This is called *text scanning*. Text scanning is applicable even if the text is updated from time to time and it does not require index structures. In general, text indexing is superior in terms of search speed, while text scanning is superior in terms of search flexibility. By convention, "pattern matching" often refers to the latter, text scanning, in a narrow sense.

In this chapter, we outline a fusion technology of data compression and pattern matching called *compressed pattern matching*. First, in Sect. 7.2, we look back on the history of this field of study. Then, in Sect. 7.3, we provide some notation and definitions that are used in the following sections. In addition, we recall grammar compression, which is the key compression scheme for compressed pattern matching. Next, in Sect. 7.4, we introduce the general framework of compressed pattern matching proposed by Kida et al. [21]. In Sects. 7.5 and 7.6, we present outlines of Repair-VF and MR-Repair, respectively, which are the results of our work in this study. Finally, we conclude in Sect. 7.7.

7.2 History of Compressed Pattern Matching Research

The technology of combining data compression and pattern matching emerged in the early 1990s. This has come to be known as the *compressed pattern matching problem* [1], which is the problem of performing pattern matching without first decompressing the compressed input text. Formally, when a text $T = t_1 t_2 \dots t_u$ (t_i is a symbol) is given in compressed form $Z = z_1 z_2 \dots z_n$ (z_i is an element of the compressed text), and pattern P is given, the problem is to find all occurrences of P in T using only Z and P. A simple method is to first decompress Z to T and then use some commonly used pattern matching algorithm. However, this approach requires $O(m + u)$ time for pattern matching in addition to the decompression time.

The optimal algorithm for the compressed pattern matching problem is one that performs pattern matching in $O(m + n)$ time in the worst case. However, it is not easy to achieve both efficient compression of text data and fast pattern matching on it. In the initial research in this field, individual pattern matching algorithms were developed for each compression method. For example, Eilam-Tzoreff and Vishkin et al. [15] proposed an algorithm for run-length compression, Gąsieniec et al. [18] and Farach Thorup et al. [16] proposed algorithms for LZ77, and Amir et al. [2] proposed an algorithm for LZW [54]. However, these algorithms tend to be complicated, and

as Manber [29] pointed out, it is questionable as to whether they are more practical than the simple method.

From the late 1990s to the early 2000s, several efficient methods for compressed pattern matching emerged [22, 38, 41]. For the first time, it was shown experimentally that these methods can perform pattern matching faster than the simple method. Furthermore, methods have appeared that can perform pattern matching faster by about the compression ratio than matching on the original text. The key is to select a compression method suitable for pattern matching even at the expense of compression ratio. In fact, Byte Pair Encoding (BPE), which was used by Shibata et al. [48] for this purpose, has a compression ratio of at most about 50% for natural language texts, while the LZ-family methods can compress the same texts to about 30% or less. However, text compression by BPE offers an advantage for pattern patching because all the codewords are fixed at 8 bits and the correspondence between each codeword and a portion of the text is relatively clear.

This caused a paradigm shift. Whereas individual pattern matching algorithms were previously developed for each data compression method, we realized that in order to increase the matching speed it would be better to develop a new data compression method suitable for pattern matching. In fact, in the 2000s, several data compression methods were proposed for this purpose.

One of the main groups of compression methods based on this idea are the compression methods proposed by Brisaboa et al. [7–9] and by Klein and Ben-Nissan [24]. These are based on a technique called *dense coding* [8]. Dense coding divides an input (natural language) text into words, and then encodes them so that the codewords become shorter in descending order of the frequency of the words. In addition, each codeword is assigned a bit pattern that has an explicit end to facilitate codeword extraction. Although dense coding offers good performance in terms of both compression ratio and pattern matching speed, some ingenuity is required to apply it to data such as DNA sequences that cannot be divided into words.

The other system is *grammar-based compression* (or *grammar compression*) [23] with fixed-length coding. This idea is an extension of BPE and can be applied even if an input text cannot be separated into words. One direct improvement of BPE is a method using a context-sensitive grammar by Maruyama et al. [34], while for compression methods based on context-free grammar we have the methods by Klein and Shapira [25] and Uemura et al. [51]. In both methods, a modified version of suffix tree [53] is used as a dictionary tree for constructing grammar.

In this chapter, for convenience, we refer to the former system as the dense coding system and the latter as the VF coding system.

In the 2010s, new data compression algorithms began to appear that achieved compression performance comparable to well-known compression tools such as Gzip and Bzip while maintaining properties suitable for pattern matching. Among the two systems described above, the VF coding system has difficulties in terms of compression rate and compression speed. Therefore, research looked into searching for and improving grammar compression, which is the basis of the VF coding system. Among this work, the algorithm RePair [26], which was proposed before the name "grammar compression" was used, has attracted attention because of its excellent

compression ratio. Yoshida and Kida et al. [56] proposes a variant of RePair, called Repair-VF, which reduces the decrease in compression ratio by suppressing unnecessary grammar rules while encoding the output using fixed length codewords. The time and space complexities required for Repair-VF are both $O(n)$ for text of length n, which is the same as the original RePair. Repair-VF realizes high-speed pattern matching on compressed text while having a good compression ratio comparable to Gzip.

Very recently, we proposed a novel variant of RePair, called MR-RePair [17], which constructs more compact grammars than RePair, particularly for highly repetitive texts. This achievement comes from an analysis of RePair. We show in [17] that the main process of RePair, that is, the step by step substitution of the most frequent symbol pairs, works within the corresponding most frequent *maximal repeats*. We then reveal in [17] the relationship between maximal repeats and grammars constructed by RePair.

7.3 Preliminaries

7.3.1 Definitions of Notation and Terms

Let Σ be an *alphabet*, that is, an ordered finite set of symbols. An element $T = t_1 \dots t_n$ of Σ^* is called a *string* or a *text*, where $|T| = n$ denotes its length. Let ε be an empty string of length 0, that is, $|\varepsilon| = 0$. We denote a concatenation of two strings $x, y \in \Sigma$ by $x \cdot y$, or xy for simplicity if no confusion occurs.

If $T = xyz$ with $x, y, z \in \Sigma^*$, then x, y, z are called a *prefix*, *substring*, and *suffix* of T, respectively. Let $T[i : j] = t_i \cdots t_j$ for any $1 \leq i \leq j \leq n$ denote a substring of T beginning at i and ending at j in T, and let $T[i] = t_i$ denote the i th symbol of T. Let $w[i : j] = \varepsilon$ if $j < i$ for simplicity.

7.3.2 Grammar Compression

A *context-free grammar* (*CFG* or simply *grammar*) G is defined as a four-tuple $G = \{V, \Sigma, S, R\}$, where V denotes an ordered finite set of variables, Σ denotes an ordered finite alphabet, R denotes a finite set of binary relations called *production rules* (or *rules*) between V and $(V \cup \Sigma)^*$, and $S \in V$ denotes a special variable called *start variable*. A production rule refers to the situation where a variable is substituted and written in the form $v \to w$, with $v \in V$ and $w \in (V \cup \Sigma)^*$. Let $X, Y \in (V \cup \Sigma)^*$. If there are $x_l, x, x_r, y \in (V \cup \Sigma)^*$ such that $X = x_l x x_r$, $Y = x_l y x_r$, and $x \to y \in R$, we write $X \to Y$, and denote the reflexive transitive closure of $\to$ as $\stackrel{*}{\Rightarrow}$. Let $val(v)$ be a string derived from v, that is, $v \stackrel{*}{\Rightarrow} val(v)$. We define grammar $\hat{G} = \{\hat{V}, \hat{\Sigma}, \hat{S}, \hat{R}\}$ as a *subgrammar* of G if $\hat{V} \subseteq V$, $\hat{\Sigma} \subseteq (V \cup \Sigma)$, and $\hat{R} \subseteq R$.

Given a text T, *grammar compression* is a method for lossless text data compression that constructs a restricted CFG uniquely deriving the text T. For G to be deterministic, the production rule for each variable $v \in V$ must be unique. In what follows, we assume that every grammar is deterministic and each production rule is $v_i \to expr_i$, where $expr_i$ is an expression either $expr_i = a$ $(a \in \Sigma)$ or $expr_i = v_{j_1} v_{j_2} \ldots v_{j_n}$ $(i > j_k$ for all $1 \leq k \leq j_n)$. To estimate the effectiveness for compression, we use the size of the constructed grammar, which is defined as the total length of the right-hand side of all production rules of the grammar.

While the problem of constructing the smallest such grammar for a given text is known to be NP-hard [10], several approximation algorithms have been proposed. One of them is RePair [26], which is an off-line grammar compression algorithm. Despite its simple scheme, RePair is known for its high compression in practice [12, 19, 52], and hence, it has been comprehensively studied. Some examples of studies on the RePair algorithm include its extension to an online algorithm [35], practical working time/space improvements [6, 47], applications to various fields [12, 27, 49], and theoretical analysis of generated grammar sizes [10, 40, 42].

7.4 Framework for Compressed Pattern Matching

A grammar compressed text can be expressed in a framework called collage systems [21]. A pattern matching algorithm on the compressed text can then be obtained as an instance of the general algorithm on the collage system. Algorithm on collage systems can be understood as an extension of the Knuth-Morris-Pratt method (KMP method) [14].

7.4.1 KMP Method

The KMP method a well-known linear-time algorithm for pattern matching on an ordinary (uncompressed) text. Its movement can be modeled as a linear automaton (KMP automaton) for a given pattern P.

For a given pattern P, a KMP automaton consists of two functions:

$$\begin{aligned} \text{goto function} g &: Q \times \Sigma \to Q \cup \{fail\}, \\ \text{failure function} f &: Q \setminus \{0\} \to Q, \end{aligned}$$

where $Q = \{0, 1, \ldots, |P|\}$ is the set of states, and *fail* is a special symbol that is not included in Q. For $j \in Q$ and $a \in \Sigma$, the goto function g returns $j + 1$ if $P[j + 1] = a$ holds, otherwise it returns *fail*. For $j = 0$, let $g(0, a) = 0$ for all $a \in \Sigma$ where $P[1] \neq a$ holds. For $j \in Q \setminus \{0\}$, the failure function f returns the maximum integer k such that $P[1 : k] = P[j - k + 1 : j]$ holds.

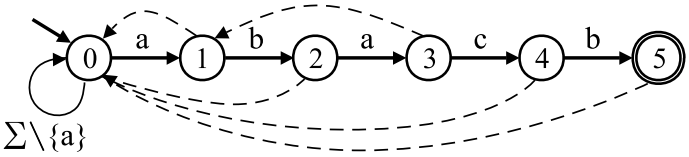

Fig. 7.1 KMP automaton for $P = \texttt{abacb}$. In this figure, each circle indicates a state, and the double circle indicates the final state. Solid arrows and dashed arrows indicate the goto function and failure function, respectively

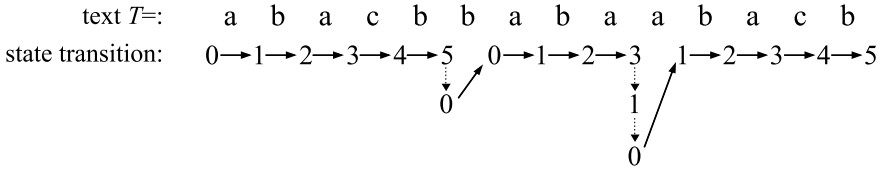

Fig. 7.2 Movement of KMP automaton. Solid arrows and dashed arrows indicate state transitions caused by the goto function and the failure function, respectively

The automaton repeats state transitions by tracing g corresponding to the characters read one by one from the input text. If g returns *fail*, then f is repeatedly called with the current state number to go back until a transition by g succeeds with the same character. When the automaton finally reaches the rightmost state, it can be judged that P has occurred.

Figure 7.1 shows the KMP automaton for pattern $P = \texttt{abacb}$. The movement of the KMP automaton in Fig. 7.1 for the text $T = \texttt{abacbbabaabacb}$ is shown in Fig. 7.2. In this example, it can be judged that P has occurred when the automaton reaches the state number 5.

To eliminate the failure function, we define the state transition function $\delta : Q \times \Sigma \to Q$ as follows:

$$\delta(j, a) = \begin{cases} g(j, a) & \text{if } g(j, a) \neq \mathit{fail}, \\ \delta(f(j), a) & \text{otherwise} \end{cases}$$

Moreover, we extend it to $Q \times \Sigma^*$ as follows:

$$\delta^*(j, \varepsilon) = j, \quad \delta^*(j, ua) = \delta(\delta^*(j, u), a),$$

where $j \in Q$, $u \in \Sigma^*$, and $a \in \Sigma$.

7.4.2 Collage System

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows: $\mathcal{D}$ is a sequence of assignments $X_1 = expr_1; X_2 = expr_2; \cdots ; X_n = expr_n$, where each X_k is a token and $expr_k$ is any of the form:

a for $a \in \Sigma \cup \{\varepsilon\}$, (*primitive assignment*)
$X_i X_j$ for $i, j < k$, (*concatenation*)
$^{[j]}X_i$ for $i < k$ and an integer j, (*prefix truncation*)
$X_i^{[j]}$ for $i < k$ and an integer j, (*suffix truncation*)
$(X_i)^j$ for $i < k$ and an integer j. (j *times repetition*)

Let the set of all tokens in $\mathcal{D}$ be denoted by $F(\mathcal{D})$. Each token represents a string obtained by evaluating the expression as it implies. Let the string represented by token $X \in F(\mathcal{D})$ be denoted by $X.u$. For example, for $X_1 = \mathtt{a}; X_2 = \mathtt{b}; X_3 = X_1 \cdot X_2; X_4 = (X_3)^3; X_5 = X_4^{[1]}$, $X_4.u = \mathtt{ababab}$ and $X_5.u = \mathtt{ababa}$. However, in this section we identify token X with the string it represents, and simply denote both by X unless confusion occurs.

Let the number of assignments in $\mathcal{D}$ be the *size* of $\mathcal{D}$, and denote it by $||\mathcal{D}||$, that is, $||\mathcal{D}|| = |F(\mathcal{D})| = n$. For a sequence $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_k}$ of tokens defined in $\mathcal{D}$, we denote by $|\mathcal{S}|$ the number k of tokens in $\mathcal{S}$.

The collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ represents the string obtained by concatenating $X_{i_1}, \ldots, X_{i_k}$. That is, $\mathcal{D}$ and $\mathcal{S}$ correspond to the dictionary and compressed text in a compression method, respectively. Both $\mathcal{D}$ and $\mathcal{S}$ can be encoded in various ways. The compression ratios therefore depend on their encoding sizes rather than $||\mathcal{D}||$ and $|\mathcal{S}|$.

A collage system is said to be *truncation-free* if $\mathcal{D}$ contains no truncation operation. A collage system is said to be *regular* if $\mathcal{D}$ contains neither truncation nor repetition operations. A regular collage system is said to be *simple* if for every assignment $X = YZ$, $|Y.u| = 1$ or $|Z.u| = 1$.

7.4.3 Pattern Matching on Collage Systems

The basic idea of pattern matching on a collage system is to simulate the movement of the KMP automaton on uncompressed text. Using the state transition function δ^* of the KMP automaton defined in Sec. 7.4.1, we define the function $Jump : Q \times F(\mathcal{D}) \to Q$ as follows:

$$Jump(j, X) = \delta^*(j, X.u).$$

The intent of *Jump* is to simulate the state transition of the original KMP automaton by jumping when it receives token X. Moreover, for any $j \in Q$ and $X \in F(\mathcal{D})$, we define the set $Output(j, X) = Occ_P(P[1 : j] \cdot X.u)$, where $Occ_P(x)$ indicates the set of all indices of occurrences of P within x.

Fig. 7.3 A matching algorithm on a collage system

Input. Collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ and pattern $P = P[1 : m]$.
Output. All positions of occurrences of P in T.
/* *Preprocessing* */
Collect the information required to calculate *Jump* and *Output*;
/* *Scanning compressed text* */
let $\mathcal{S} = X_{i_1} X_{i_2} \ldots X_{i_n}$.
$\ell := 0$; $state := 0$;
for $k := 1$ **to** n **do begin**
  for each $p \in Output(state, X_{i_k})$ **do**
    pattern P occurs at $\ell + p - m + 1$;
  $state = Jump(state, X_{i_k})$;
  $\ell := \ell + |X_{i_k}|$
end

For a given collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ representing text T and for a given pattern P, the pattern matching algorithm preprocesses the information required to calculate *Jump* and *Output* from $\mathcal{D}$, and then performs matching while scanning a sequence of tokens in $\mathcal{S}$ one by one from the head (Fig. 7.3).

From the results of [21], the following theorem is obtained.

Theorem 1 (Theorem 3 of [21]) *For a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$, the compressed pattern matching problem can be solved in $O(||\mathcal{D}|| + |\mathcal{S}| + m^2 + R)$ time using $O(||\mathcal{D}|| + m^2)$ space if $\langle \mathcal{D}, \mathcal{S} \rangle$ is regular, where R is the number of occurrences of pattern P in the text represented by $\langle \mathcal{D}, \mathcal{S} \rangle$.*

This theorem applies to both RePair and Repair-VF because texts compressed by these can be described by regular collage systems.

7.5 Repair-VF

This section first introduces RePair and then gives an outline of Repair-VF. Repair-VF has a structure that combines RePair with a fixed-length coding. Please refer to the literature [56] for the details of Repair-VF and experimental results for its performance.

7.5.1 RePair

RePair is a grammar compression algorithm that was proposed by Larsson and Moffat [26]. For input text T, let $G = \{V, \Sigma, S, R\}$ be the grammar constructed by RePair. The RePair procedure can then be described by the following steps:

Step 1. Replace each symbol $a \in \Sigma$ with a new variable v_a and add $v_a \to a$ to R.
Step 2. Find the most frequent pair p in T.

	a	b	r	a	c	a	d	a	b	r	a
$v_\alpha \rightarrow \alpha$ (α = a,b,r,c,d)	v_a	v_b	v_r	v_a	v_c	v_a	v_d	v_a	v_b	v_r	v_a
$v_1 \rightarrow v_a v_b$	v_1		v_r	v_a	v_c	v_a	v_d	v_1		v_r	v_a
$v_2 \rightarrow v_1 v_r$	v_2			v_a	v_c	v_a	v_d	v_2			v_a
$v_3 \rightarrow v_2 v_a$	v_3				v_c	v_a	v_d	v_3			
$S \rightarrow v_3 v_c v_a v_d v_3$	S										

Fig. 7.4 Example of the grammar generation process of RePair for T = abracadabra. The generated grammar is $\{\{v_a, v_b, v_r, v_c, v_1, v_2, v_3, S\}, \{a,b,r,c,d\}, S, \{v_a \rightarrow a, v_b \rightarrow b, v_r \rightarrow r, v_c \rightarrow c, v_d \rightarrow d, v_1 \rightarrow v_a v_b, v_2 \rightarrow v_1 v_c, v_3 \rightarrow v_2 v_d, S \rightarrow v_3 v_c v_a v_d v_3\}\}$ with a size of 16

Step 3. Replace every occurrence (or as many occurrences as possible if p is a pair consisting of the same symbol) of p with a new variable v, and then, add $v \rightarrow p$ to R.

Step 4. Re-evaluate the frequencies of pairs for the updated text generated in Step 3. If the maximum frequency is 1, add $S \rightarrow$ (current text$T$) to R, and terminate. Otherwise, return to Step 2.

Figure 7.4 illustrates an example of the grammar generation process of RePair.

The following theorem relates to the performance of RePair shown by Larsson and Moffat [26].

Theorem 2 ([26]) *RePair works in $O(n)$ expected time and $5n + 4k^2 + 4k' + \lceil\sqrt{n+1}\rceil - 1$ words of space, where n is the length of the source text, k denotes the cardinality of the source alphabet, and k' denotes the cardinality of the final dictionary.*

7.5.2 Outline of Repair-VF

The original RePair encodes the rules in R excluding S using Elias gamma coding, that is, each codeword has a variable length, whereas Repair-VF uses a fixed-length code. The right side of S corresponds to the compressed text, and is converted to a sequence of fixed-length codewords of the rules in S.

Consider the number of fixed-length coded rules. In the process of Step 1 of RePair, $|\Sigma|$ rules are created. In addition, the process of Step 3 of RePair replaces the most frequent pair and at the same time adds one rule to R. Let s be the number of rules which are added to R in Step 3. The total number of rules is then $|\Sigma| + s$, and thus each symbol can be fixed-length encoded with $\lceil\log(|\Sigma| + s)\rceil$ bits. The information about Σ can be restored from the rules added in Step 3, so there is no need to explicitly save it. Therefore, only the rules added in Step 3 and the right side of S added last in Step 4 need to be saved. In the former, the right side of each rule consists of two symbols, so the total number of symbols to be saved is $2s$. Since the latter depends on the input text T, let n be the length of T. The number of bits of the output compressed data is then

$$(2s + n)\lceil \log(|\Sigma| + s) \rceil \tag{7.1}$$

bits in total.

We want to find the best s that minimizes the total number of output bits. Note that the final output tends to be smaller as s increases up to some point. In RePair, Step 3 is repeated until the frequency of the most frequent pair becomes 1. In the case of using a fixed-length code as above, this increases useless rules. Increasing the number of rules can increase the bit length per symbol, resulting in a longer final bit length. We can eliminate the waste by terminating the process in the middle. However, it is difficult to determine on the fly the s at which the output becomes minimal. Even after the first time the output size increases, the length of S may become shorter by continuing Step 3, and the output size may decrease again.

Therefore, during the processing of RePair, we record the minimum value of the output size and the corresponding s by calculating Equation (7.1) every time a rule is added in Step 3. Note that the calculation of Equation (7.1) does not require actual coding or outputting since a fixed-length code is used. Finally, when S is output in Step 4, we can obtain the smallest output by outputting while expanding the rules added after the best s.

This is *Repair-VF* (called Repair-VF-best in the original paper). The suffix "VF" comes from an abbreviation for variable-to-fixed length coding (VF coding). For the input text T, each rule of the output grammar G corresponds to a substring of T, and the right-hand side of S can be regarded as the variable length factorization of T. Thus, Repair-VF can be viewed as a VF coding from the viewpoint of information source coding.

7.6 MR-Repair

In this section we outline MR-Repair, which is a method to reduce the output grammar size by focusing on the relationship between RePair and maximal repeats. Please refer to the literature [17] for the details of MR-Repair and experimental results for its performance.

7.6.1 Maximal Repeats

Let s be a substring of text T. If the frequency of s is greater than 1, s is called a *repeat*. A *left* (or *right*) *extension* of s is any substring of T in the form ws (or sw), where $w \in \Sigma^*$. We define s as a *left* (or *right*) *maximal* if left (or right) extensions of s occur a strictly lower number of times in T than s. Accordingly, s is a maximal repeat of T if s is both left and right maximal. In this paper, we only consider strings with a length of more than 1 as maximal repeats. For example, the substring `abra` of $T =$ `abracadabra` is a maximal repeat, whereas `br` is not.

The following theorem describes an essential property of RePair, that is, RePair recursively replaces the most frequent maximal repeats.

Theorem 3 (Theorem 1 of [17]) *Let T be a given text, under the condition that every most frequent maximal repeat of T does not appear overlapping itself. Let f be the frequency of the most frequent pairs of T, and t be a text obtained after all pairs with frequency f in T are replaced by variables. There is then a text s such that s is obtained after all maximal repeats with frequency f in T are replaced by variables, and s and t are isomorphic to each other.*

7.6.2 MR Order

According to Theorem 1 of [17], if there is just one most frequent maximal repeat in the current text, then RePair replaces all occurrences of it step by step. However, a problem arises if there are two or more most frequent maximal repeats, with some of them overlapping. In this case, the selection order of pairs (of course, they are most frequent) affects the priority of maximal repeats. We call this order of selecting (summarizing) maximal repeats the *maximal repeat selection order* (or simply *MR-order*). Note that the selection order of pairs actually depends on the implementation of RePair. If there are several distinct most frequent pairs with overlaps, RePair constructs grammars with different sizes according to the selection order of the pairs.

However, the following theorem states that the MR-order rather than the replacement order of pairs determines the size of the grammar generated by RePair.

Theorem 4 (Theorem 2 of [17]) *The sizes of grammars generated by RePair are the same if they are generated in the same MR-order.*

7.6.3 Algorithm

The main strategy of the proposed method is to recursively replace the most frequent maximal repeats instead of the most frequent pairs.

Definition 1 (*Definition 3 of* [17]) For an input text T, let $G = \{V, \Sigma, S, R\}$ be the grammar generated by MR-Repair. MR-Repair constructs G through the following steps:

Step 1. Replace each symbol $a \in \Sigma$ with a new variable v a and add $v_a \rightarrow a$ to R.

Step 2. Find the most frequent maximal repeat r in T.

Step 3. Check if $|r| > 2$ and $r[1] = r[|r|]$, and if so, use $r[1 : |r| - 1]$ instead of r in Step 4.

	a	b	r	a	c	a	d	a	b	r	a
$v_\alpha \rightarrow \alpha$ (α = a,b,r,c,d)	v_a	v_b	v_r	v_a	v_c	v_a	v_d	v_a	v_b	v_r	v_a
$v_1 \rightarrow v_\mathrm{a} v_\mathrm{b} v_\mathrm{r}$	v_1			v_a	v_c	v_a	v_d	v_1			v_a
$v_2 \rightarrow v_1 v_\mathrm{a}$	v_2				v_c	v_a	v_d	v_2			
$S \rightarrow v_2 v_\mathrm{c} v_\mathrm{a} v_\mathrm{d} v_2$	S										

Fig. 7.5 Example of the grammar generation process of MR-Repair for T = abracadabra. The generated grammar is $\{\{v_a, v_b, v_r, v_c, v_d, v_1, S\}, \{a, b, r, c, d\}, S, \{v_a \rightarrow a, v_b \rightarrow b, v_r \rightarrow r, v_c \rightarrow c, v_d \rightarrow d, v_1 \rightarrow v_a v_b v_r, v_2 \rightarrow v_1 v_a, S \rightarrow v_2 v_c v_a v_d v_2\}\}$ with a size of 15

Step 4. Replace every occurrence of r with a new variable v and then add $v \rightarrow r$ to R.

Step 5. Re-evaluate the frequencies of maximal repeats for the updated text generated in Step 4. If the maximum frequency is 1, add $S \rightarrow$ (current text) to R and terminate. Otherwise, return to Step 2.

Figure 7.5 shows an example of the grammar generation process of MR-Repair. As shown in this figure, the size of the grammar generated by MR-Repair is smaller than that generated by RePair shown in Figure reffig:repair.

Theorem 5 (Theorem 5 of [17]) *Assume that RePair and MR-Repair work based on the same MR-order for a given text. Let g_{rp} and g_{mr} be the sizes of the grammars generated by RePair and MR-Repair, respectively. Then, $\frac{1}{2} g_{rp} < g_{mr} \leq g_{rp}$ holds.*

7.7 Conclusion

In this chapter, we outlined research on compressed pattern matching and showed that we can speed up pattern matching by selecting a suitable compression method. This has led to the development of compression methods that are useful for pattern matching. Whereas the initially developed compression methods had low compression performance, Repair-VF [56] achieves both a good compression rate and good matching speed by combining advanced grammar compression with fixed-length code. Collage systems [21] provide a unified algorithm for compressed pattern matching, allowing us to obtain an efficient pattern matching algorithm for grammar compression as an instance of the unified algorithm.

Since proposing Repair-VF, we have proposed several improvements for it. LT-Repair [35] improves RePair processing semi-online by adding the constraint called the *left-tall condition* to its grammar. This makes it possible to efficiently compress large-scale text data with small memory.

MR-Repair [17], which we have recently proposed, is a method that reduces the output grammar size by focusing on the relationship between RePair and maximal repeats. Although heuristic improvements [4, 20, 28, 36] focusing on non-maximal repetitive substrings have previously been proposed, MR-Repair is superior because

it has been proven to generate theoretically smaller grammar than the original RePair. A topic for future work is to see whether compressed pattern matching using these methods can be performed efficiently.

In the present work, we mainly explained the compressed pattern matching problem based on text scanning. However, the *succinct index* technology which combines text index and data compression was also established in 2000. This is an indexing technology that utilizes a succinct data structure that can solve query processing with a small space of almost the information-theoretic lower bound. Succinct index has an excellent property that allows full-text searching while compressing a target text smaller than the original text. For details of this technology, refer to the excellent book by Navarro [37].

In terms of online grammar compression methods, there exists FOLCA proposed by Maruyama et al. [33] and its improvement SOLCA proposed by Takabatake et al. [50]. FOLCA is based on a string factorization called edit-sensitive parsing. It performs factorization and grammar generation in parallel while reading an input text sequentially from the beginning. It is known that *straight line programs* (restricted CFGs) generated by FOLCA can be used as index structures [5].

In recent years, Martinez et al. [31] proposed a novel compression method called Marlin and an improvement of it [32]. These methods achieve both decompression at ultrahigh-speed and good performance in terms of compression ratio. If we can decompress compressed data at sufficiently high speed, we can perform pattern matching efficiently even if it is performed after decompressing the data. Comparative studies on these approaches are also left for future work.

References

1. A. Amir, G. Benson, Efficient two-dimensional compressed matching, in *Proc. Data Compression Conference*, p. 279 (1992)
2. A. Amir, G. Benson, M. Farach, Let sleeping files lie: pattern matching in Z-compressed files. J. Comput. Syst. Sci. **52**, 299–307 (1996)
3. A. Apostolico, Z. Galil, *Pattern Matching Algorithms* (Oxford University Press, 1997)
4. A. Apostolico, S. Lonardi, Off-line compression by greedy textual substitution. Proc. IEEE **88**(11), 1733–1744 (2000)
5. D. Belazzougui, P. Cording, S. Puglisi, Y. Tabei, Access, rank, and select in grammar-compressed strings, in *Algorithms—ESA 2015, LNCS*, vol. 9294 (Springer, 2015), pp. 142–154
6. P. Bille, I.L. Gørtz, N. Prezza, *Space-Efficient Re-pair Compression* (2017), pp. 171–180
7. N. Brisaboa, A. Fariña, J. López Rodríguez, G. Navarro, E. Lopez, A new searchable variable-to-variable compressor. Data Compression Conf. **DCC2010**, 199–208 (2010)
8. N. Brisaboa, E. Iglesias, G. Navarro, J. Paramá, An efficient compression code for text databases. Eur. Conf. Inform. Retrieval (ECIR'03) **2633**, 468–481 (2003)
9. N. Brisaboa, G. Navarro, M. Esteller, (S,C)-dense coding: An optimized compression code for natural language text databases, in *String Processing and Information Retrieval (SPIRE2003), LNCS*, vol. 2857 (2003), pp. 122–136
10. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem. Inform. Theory, IEEE Trans. **51**, 2554–2576 (2005)
11. C. Charras, T. Lecroq, *Handbook of Exact String Matching Algorithms* (College Publications, 2004)

12. F. Claude, G. Navarro, Fast and compact web graph representations. ACM Trans. Web **4**(4) (2010). https://doi.org/10.1145/1841909.1841913
13. M. Crochemore, C. Hancart, T. Lecroq, *Algorithms on Strings* (Cambridge University Press, 2007)
14. M. Crochemore, W. Rytter, *Jewels of Stringology* (World Scientific, 2002). https://doi.org/10.1142/4838
15. T. Eilam-Tzoreff, U. Vishkin, Matching patterns in strings subject to multi-linear transformations. Theor. Comput. Sci. **60**(3), 231–254 (1988)
16. M. Farach, M. Thorup, String-matching in Lempel-Ziv compressed strings. Algorithmica **20**(4), 388–404 (1998). ((previous version in STOC'95))
17. I. Furuya, T. Takagi, Y. Nakashima, S. Inenaga, H. Bannai, T. Kida, MR-RePair: grammar compression based on maximal repeats, in *Data Compression Conference (DCC2019)* (IEEE Computer Society, 2019), pp. 508–517
18. L. Gąsieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for Lempel-Ziv encoding, in *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, LNCS*, vol. 1097, (Springer, 1996), pp. 392–403
19. R. González, G. Navarro, Compressed text indexes with fast locate, in *Combinatorial Pattern Matching*. ed. by B. Ma, K. Zhang (Springer, Berlin Heidelberg, Berlin, Heidelberg, 2007), pp. 216–227
20. S. Inenaga, T. Funamoto, M. Takeda, A. Shinohara, Linear-time off-line text compression by longest-first substitution, in *String Processing and Information Retrieval. SPIRE 2003, LNCS*, vol. 2857 (Springer, 2003), pp. 137–152
21. T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa, Collage system: a unifying framework for compressed pattern matching. Theor. Comput. Sci. **298**(1), 253–272 (2003). https://doi.org/10.1016/S0304-3975(02)00426-7
22. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa, Multiple pattern matching in LZW compressed text. J. Discrete Algor. **1**(1), 133–158 (2000). (previous version in DCC'98 and CPM'99)
23. J.C. Kieffer, E. Yang, Grammar-based codes: a new class of universal lossless source codes. IEEE Trans. Inform. Theory **46**(3), 737–754 (2000)
24. S. Klein, M. Ben-Nissan, Using fibonacci compression codes as alternatives to dense codes. Data Compression Conf. **DCC2008**, 472–481 (2008)
25. S. Klein, D. Shapira, Improved variable-to-fixed length codes, in *String Processing and Information Retrieval (SPIRE 2008), LNCS*, vol. 5280, (Springer, 2008), pp. 39–50
26. N.J. Larsson, A. Moffat, Offline dictionary-based compression, in *Data Compression Conference (DCC'99)*, (IEEE Computer Society, 1999), pp. 296–305
27. M. Lohrey, S. Maneth, R. Mennicke, Xml tree structure compression using re-pair. Inform. Syst. **38**(8), 1150–1167 (2013)
28. M. Gańczorz, A. Jeż, Improvements on repair grammar compressor. Data Compression Conf. **DCC2017**, 181–190 (2017)
29. U. Manber, A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inform. Syst. **15**(2), 124–136 (1997). (previous version in CPM'94)
30. C.D. Manning, P. Raghavan, H. Schuetze, *Introduction to Information Retrieval* (Cambridge University Press, 2008)
31. M. Martinez, M. Haurilet, R. Stiefelhagen, J. Serra-Sagristà, Marlin: a high throughput variable-to-fixed codec using plurally parsable dictionaries, in *Data Compression Conference (DCC2017)*, (IEEE Computer Society, 2017), pp. 161–170
32. M. Martinez, K. Sandfort, D. Dubé, J. Serra-Sagristà, Improving marlin's compression ratio with partially overlapping codewords, in *Data Compression Conference (DCC2018)* (IEEE Computer Society, 2018), pp. 325–334
33. S. Maruyama, Y. Tabei, Fully online grammar compression in constant space, in *Data Compression Conference (DCC2014)*, (IEEE Computer Society, 2014), pp. 173–182
34. S. Maruyama, Y. Tanaka, H. Sakamoto, M. Takeda, Context-sensitive grammar transform: compression and pattern matching. IEICE. Trans. **93-D**, 219–226 (2010). https://doi.org/10.1587/transinf.E93.D.219

35. T. Masaki, T. Kida, Online grammar transformation based on re-pair algorithm, in *Data Compression Conference (DCC2016)*, (IEEE Computer Society, 2016), pp. 349–358
36. R. Nakamura, H. Bannai, S. Inenaga, M. Takeda, Simple linear-time off-line text compression by longest-first substitution, in *Data Compression Conference (DCC'07)*, (IEEE Computer Society, 2007), pp. 123–132
37. G. Navarro, *Compact Data Structures: A Practical Approach*, 1st edn. (Cambridge University Press, USA, 2016)
38. G. Navarro, M. Raffinot, A general practical approach to pattern matching over Ziv-Lempel compressed text. Combinat. Pattern Matching (CPM'99), LNCS **1645**, 14–36 (1999)
39. G. Navarro, M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences* (Cambridge University Press, 2002)
40. G. Navarro, L. Russo, *Repair Achieves High-Order Entropy* (2008), pp. 537–537
41. G. Navarro, J. Tarhio, Boyer-moore string matching over Ziv-Lempel compressed text, in *Proceedings of the CPM'2000, LNCS*, vol. 1848, (Springer, 2000), pp. 166–180
42. C. Ochoa, G. Navarro, Re-pair and all irreducible grammars are upper bounded by high-order empirical entropy. IEEE Trans. Inform. Theory **65**(5), 3160–3164 (2019)
43. D. Salomon, *Data Compression: The Complete Reference*, 4th edn. (Springer, 2006)
44. D. Salomon, G. Motta, *Handbook of Data Compression*, 5th edn. (Springer, 2009)
45. K. Sayood, *Lossless Compression Handbook* (Academic Press, 2002)
46. K. Sayood, *Introduction to Data Compression*, 4th edn. (Morgan Kaufmann, 2012)
47. K. Sekine, H. Sasakawa, S. Yoshida, T. Kida, *Adaptive Dictionary Sharing Method for Repair Algorithm*, (2014), p. 425
48. Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, S. Arikawa, Speeding up pattern matching by text compression, in *Proceedings of the 4th Italian Conference on Algorithms and Complexity, LNCS*, vol. 1767, (Springer, 2000), pp. 306–315
49. Y. Tabei, H. Saigo, Y. Yamanishi, S.J. Puglisi, *Scalable partial least squares regression on grammar-compressed data matrices*, vol. KDD '16, (Association for Computing Machinery, New York, NY, USA, 2016), pp. 1875–1884. (https://doi.org/10.1145/2939672.2939864)
50. Y. Takabatake, I. Tomohiro, H. Sakamoto, A space-optimal grammar compression, in *25th Annual European Symposium on Algorithms (ESA2017), LIPIcs*, vol. 87, (Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017), pp. 67:1–67:15
51. T. Uemura, S. Yoshida, T. Kida, T. Asai, S. Okamoto, Training parse trees for efficient vf coding, in *String Processing and Information Retrieval (SPIRE 2010), LNCS*, vol. 6393, (Springer, 2010), pp. 179–184
52. R. Wan, *Browsing and Searching Compressed Documents* (The University of Melbourne, Melbourne, Australia, 2003). (Ph.D. thesis)
53. P. Weiner, *Linear Pattern Matching Algorithm* (1973), pp. 1–11
54. T.A. Welch, A technique for high performance data compression. IEEE Comput. **17**, 8–19 (1984)
55. I.H. Witten, A. Moffat, T.C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd edn. (Morgan Kaufmann, 1999)
56. S. Yoshida, T. Kida, A variable-length-to-fixed-length coding method using a Re-Pair algorithm. IPSJ Online Trans. **6**, 121–127 (2013)